Balancing Speed and Priority

Crafting Embedded Networks for Diverse Real-Time Communication Demands

12-SEP-2023

by Olaf Pfeiffer

A technology guide from

Jointly published by

Embedded Systems Academy, Inc.
111 N. Market Street, Suite 300
San Jose, CA 95113, USA

Embedded Systems Academy GmbH
Bahnhofstraße 17
30890 Barsinghausen, Germany

Limitation of Liability

Neither Embedded Systems Academy (EmSA) nor its authorized dealer(s) shall be liable for any defect, indirect, incidental, special, or consequential damages, whether in an action in contract or tort (including negligence and strict liability), such as, but not limited to, loss of anticipated profits or benefits resulting from the use of the information or software provided in this publication or any breach of any warranty, even if EmSA or its authorized dealer(s) has been advised of the possibilities of such damages.

The information presented in this publication is believed to be accurate. Responsibility for errors, omission of information, or consequences resulting from the use of this information cannot be assumed by EmSA. EmSA retains all rights to make changes to this publication or software associated with it at any time without notice.

Table of Contents

Edition

This is a combined PDF of the original article series published on linkedin:
https://www.linkedin.com/pulse/balancing-speed-priority-embedded-systems-academy

Introduction

When looking at articles, blogs, and posts related to real-time capable communication, they often focus on selected specific details of how "the best" can be achieved from a certain aspect of an embedded communication system like CAN, CANopen, or real-time Ethernet-based communication. It is crucial to consider how these specific details apply to a broader range of applications and their unique requirements. Many readers of such articles may question whether they are implementing these features correctly, leading to uncertainty. What I often miss is setting it all in perspective. If the required responsiveness of your system is in the area of 100ms, then you do not need to review in detail every cause adding a delay of a single millisecond or less.

To give an example, in CAN communication, collisions are resolved by prioritization. However, without collision, even the lowest priority frame gets immediate network access. Therefore, if your system only has a busload of 50% or less and some mechanisms are in place that no device can produce back-to-back high-priority traffic, then discussions about optimizing priorities or managing software handlers by priority may become purely theoretical, without significant practical application.

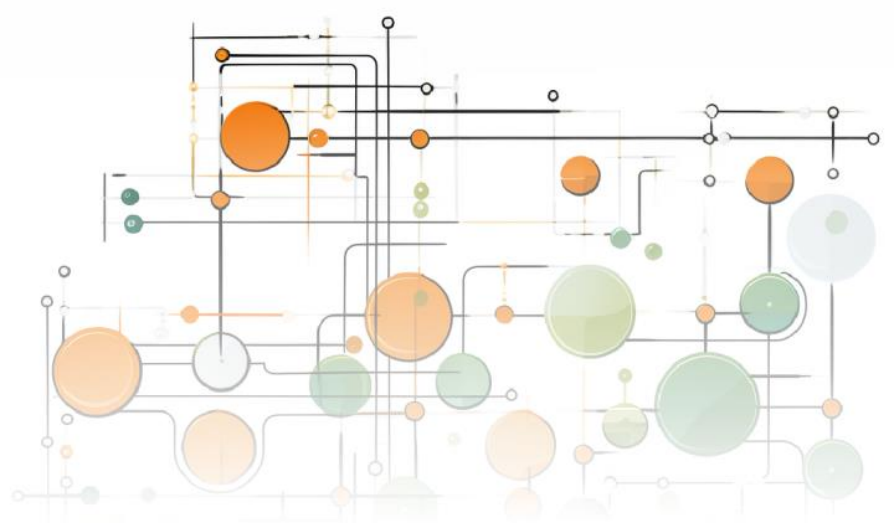
A word on 'safety' and 'security': The importance of these two aspects is increasing in a variety of applications. Adherence to specific safety standards, if required, introduces a whole new layer of complexity and consideration. The details of these standards and the intricate procedures they require can be extensive. However, to maintain the scope and length of this article, these subjects will not be explored in-depth, and the focus will remain primarily on the subject of real-time processing. In the context of timing behavior, it is essential to recognize that if your signals require safety or security measures or both, additional metadata will be necessary to safeguard the original signal data. This may include redundant information, counters, timestamps, and various cryptographic checksums.

With this series of articles, I am taking a "step back" to first find the right perspective. I start with "Part I: The Clock is Ticking: Selecting the Right Real-Time Timeframe" to review an application's requirement – to get an idea of the "ballpark" we are operating within.

In "Part II: The Demands of Real-Time Communication Systems" we look at the different timeframes required by different applications and review what this means for the communication system used.

In "Part III: The Temporal Dynamics of CAN-Based Systems," I apply our findings to CAN and CANopen, giving recommendations on "how to use" (configure) the communications to meet the demands found earlier.

The last article "Part IV: From Theory to Practice: CANopen Source Code Configuration" shows which optimization options are typically available when working with CANopen source code, here, our own Micro CANopen Plus.



1 The Clock is Ticking: Selecting the Right Real-Time Timeframe

In the world of embedded systems, real-time applications occupy a crucial niche. These applications are characterized by their requirement to process inputs and produce outputs within a specific timeframe. The accuracy of the results they provide depends not only on their logical correctness but also on the precise timing of their responses. As these systems interact with the physical world, the stakes can be high, often involving human safety, product quality, or efficient system operation. Therefore, the responsiveness of these applications becomes a basic aspect of their design.

However, “within a specific timeframe” can be very different depending on the application. For the rudder and thrust control of a large ship, this might be a second or more. For a high-speed sorting and packaging unit in a cookie factory, it might be single milliseconds. And these two cases already show nicely the different demands regarding safety: The “slow” commands in the ship need to be much more reliable (or safer) than those commands sorting cookies.

As you can imagine, the specific challenges of implementing real-time applications often depend on the communication channels involved. Are the inputs and outputs directly connected to the main processing unit, or is an embedded communication network necessary?

As applications tend to grow more complex and geographically distributed, it is impractical to have direct connections to every input and output. Instead, many real-time systems rely on remote connections. Sensors, actuators, and other devices might be located far from the central processing unit, making some form of communication between them necessary. Often, this also means that data has to be transmitted twice within the required timeframe: inputs from sensors to the processing unit and secondly the processing units' outputs to the actuators.

All of this brings additional challenges and considerations: Communication channels introduce delays, or potential data corruption or loss. Designers of real-time systems must now account for these factors, ensuring that the communication methods used can still meet the system's real-time requirements. In addition, these systems must now be able to handle multiple, often simultaneous, data streams and manage the prioritization of these streams based on their urgency and importance.

The increasing sophistication and requirements of real-time applications, coupled with the growing distance between the processing unit and input/output devices, have made the design of real-time systems a multi-faceted and challenging endeavor. Such a development demands a deeper understanding of communication protocols, network topologies, and error handling mechanisms. Only by addressing all these factors can we ensure that real-time systems continue to meet the stringent demands placed upon them.

Before diving into the design process, the first and most crucial question is determining the required timeframe for a specific application. Are we talking about seconds? Hundreds of a second? Or even milliseconds? Once a system has been fully

designed and developed, shortening the timeframe might not be possible, as many design decisions would have been based on the initial timeframe estimate. After you've established a desired timeframe for the real-time responsiveness of the system, I recommend taking some extra time to review it thoroughly. Consider having your boss, customer, or partners sign off on it, as making changes to the established timeframe later on can be costly.

If your application requires that the “the entire input to output” to be included into the calculation, then you have multiple times to add up: processing time in the input sensor to collect the input and preparing it for transmission, transmission delay, processing time in the main processor: receiving the inputs (waiting for others?), processing them and preparing for transmission to the outputs and on the outputs the processing delay of receiving the data and actually applying it.

In the following, let’s review some application examples sorted by the required response times:

1.1 Apps with Response Times Beyond Seconds

For applications operating in timeframes of single or multiple seconds, the systems often don't require special precautions. This is because the delay tolerance of these applications is significantly larger than the typical delays introduced by communication protocols. Interestingly, even when the control code is executed on slower non-real-time operating systems, timely operation is achievable. Challenges may arise if the operating system is tasked with excessive concurrent operations, but these situations are generally exceptions rather than the norm. Nevertheless, do not underestimate the consequences: even if in your application the realtime timeframe is 1s – what exactly will happen if that 1s is not met? Is that just annoying – or will something get damaged – or does the data even need to be ‘safe’ as otherwise some serious damage or even deaths could occur?

Solar Tracking of Solar Panels: Solar panels with tracking capabilities adjust according to the sun's position. Delays of seconds to minutes are typical in this application, ensuring optimal energy capture even with occasional control delays.

HVAC Systems: Heating, Ventilation, and Air Conditioning systems often incorporate sensors to modulate temperature and air quality. While immediate adjustments are beneficial, a delay of several seconds is generally well within the acceptable range.

Mining Equipment: In mining operations, large machinery such as conveyors and large-scale excavators require multiple seconds to start or stop. Given the scale, a delay of a second in system response can be acceptable, especially for non-critical adjustments. However, safety-critical functions like an emergency shut-off will have more stringent requirements.

Maritime Applications: Given the relatively slow movement dynamics of large maritime vessels, a second of delay for data processing and navigation can be acceptable.

Sub-Sea Operations: In deep-sea systems, reliability stands as the foremost priority. While managing seabed operations—from pipeline control to equipment adjustments—commands may take multiple seconds to reach their destination and cause the desired action.

1.2 Apps with Response Times of 100ms

In many scenarios, especially those centered around human-machine interaction, response times in the ballpark of 100 milliseconds are crucial. This range is rooted in the fundamental limits of human perception and reaction. When a system responds within this timeframe, the interaction feels nearly instantaneous to the user, promoting a sense of seamless control and real-time feedback. Given that the average human reaction time to visual stimuli is greater than 100ms, systems that operate within a 100ms timeframe are within the range to feel immediate and intuitive. To achieve these response times, you generally don't need to take any special measures regarding your communication channel. Even at relatively slow communication speeds like 100kbps this can be reached.

Vehicle Instrumentation and Controls: In a variety of human-controlled vehicles, such as cars, forklifts, cranes, and agricultural vehicles, a myriad of displays and controls—from touchscreens to dials—rely on swift feedback. This ensures the operator remains informed and in control. Sending controls via switches or joysticks, or receiving real-time feedback from sensors, all need to occur within this timeframe.

Industrial Machine Interfaces: Operators at manufacturing plants interact with complex machinery through control panels. Quick feedback is essential, ensuring the user's commands translate to machine actions almost instantly, which in turn enhances operational safety and efficiency. Where it takes longer to activate a command, some immediate visual feedback should be provided to signal the operator that the selected function is now about to be executed.

Medical Equipment: Devices such as patient monitors and specific diagnostic tools require timely feedback when healthcare professionals adjust settings or input commands. This prompt response ensures both patient safety and the confidence of healthcare professionals.

1.3 Apps with Response Times of 10ms

For applications demanding a response time around 10 milliseconds, precision is imperative. These timings significantly surpass the boundaries of human perception, resulting in systems often responding or adjusting even before a human can register the event. Consequently, the foundational systems must operate with unparalleled efficiency and consistency. Realizing these rigorous timings demands detailed planning, balance between speed and priority, but potentially also go deep into the software layers, including drivers and firmware, that process the data. With precise optimization, these systems exhibit the ability to react promptly, reinforcing safety, preserving functionality, and assuring peak performance.

Driver Assist Systems: Advanced Driver Assistance Systems like traction control, lane-keep assist, and anti-lock brakes are paramount in delivering quick responses. These systems sense and react to instantaneous shifts in vehicle dynamics, often in situations where any delay could lead to potential accidents.

Industrial Robotics: In state-of-the-art manufacturing setups, robotic arms and their allied machinery are tasked with instantaneous adjustments. Such promptness ensures meticulous precision, safeguards the sanctity of the production process, and curtails errors.

Emergency Shut-Off Systems: In various control settings, the quick actuation of emergency shut-off systems is crucial. Whether responding to machinery malfunctions, hazardous leaks, or any unpredictable scenario, the swift activation of these systems can prevent significant damage, financial losses, and more importantly, protect human lives.

1.4 Apps with Response Times of single Milliseconds

For applications that demand response times in the order of single milliseconds, the capabilities of several communication networks are stretched to their limits. Keep in mind that this is not about total throughput (typically only a few bytes are exchanged here) but get these bytes to the destination quickly. Achieving such rapid reactions requires a review of every facet of the system—from the configuration of the network to the underlying code—to be optimized. When getting into such demanding requirements, a comprehensive evaluation should be conducted

to determine if the chosen communication protocol is indeed the most suitable solution or if other solutions are available to complete the tasks at hand.

High-Speed Motion Control: In specialized industrial setups, machinery requires instantaneous adjustments based on rapid feedback loops. Such applications could involve fine-tuning motor speeds, swiftly actuating valves, or modulating high-speed actuators in real-time.

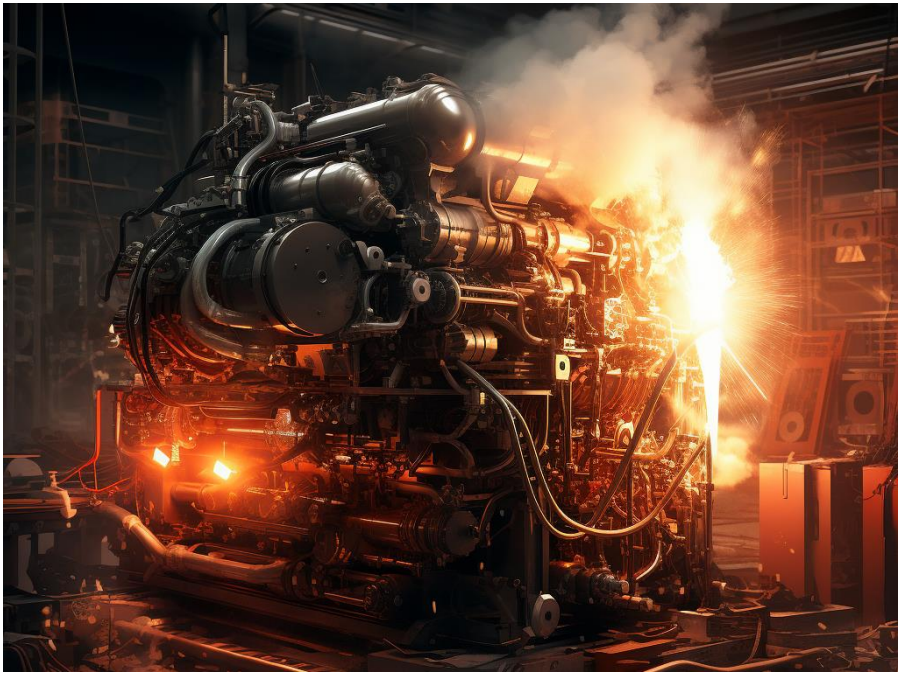
Advanced Robotics: Especially prevalent in high-precision tasks, these robots might be involved in operations like placing delicate electronic components onto a PCB at accelerated speeds, where the slightest delay can lead to significant errors.

Airbag Deployment: In vehicular safety systems, the time between detecting a potential crash and deploying an airbag can be mere milliseconds. Such a rapid response is crucial to ensure the safety of the vehicle's occupants, where every millisecond counts towards mitigating injury.

1.5 Conclusion Part I and Outlook Part II

As we have seen in this first part of our series, applications across various sectors have different response time requirements, ranging from seconds to mere milliseconds. The ability of a communication system to meet these needs is critical to achieving optimal performance and efficiency.

However, understanding these response time requirements is only one part of the puzzle. In the upcoming second part of this series, I will go deeper into the specific demands placed on a communication system to meet requirements for real-time capable communication. We will explore the technical aspects that impact communication speed, latency, and arbitration, including considerations such as network architecture, bandwidth, and data processing capabilities. Furthermore, we will examine the trade-offs and compromises that must be made when selecting a communication system that strikes a balance between speed, complexity, and cost.



2 The Demands of Real-Time Communication Systems

The ever-increasing complexity and demands of modern real-time applications necessitate robust and reliable communication systems. As established in the first part of this series, these applications span a wide spectrum of response time requirements, from seconds to milliseconds, and their success is often contingent on the precise timing of their responses. Consequently, the chosen communication system must be capable of meeting these stringent timing constraints. However, achieving the desired real-time capabilities is not the sole consideration. In many cases, these systems also need to ensure the safety of users, equipment, and the surrounding environment. Additionally, given the growing threat landscape, ensuring the security of these communication systems has become equally critical. Balancing these requirements—real-time responsiveness, safety, and security—is a multifaceted challenge.

In this second part of our series, we investigate the specific attributes and considerations that make a communication system capable of fulfilling these demands. Over time, the demands on real-time communication systems have evolved and

become more stringent. In the early days, the primary focus was on achieving real-time requirements with a reasonable level of reliability. It was often deemed sufficient if the system could process and transmit data within the specified timeframes, even if occasional errors occurred. As technology advanced and systems became more sophisticated, the need for safety became apparent. "Somewhat reliable" was no longer adequate, especially for applications where human safety, product quality, or system operation was at stake. To address these concerns, specific protocols were developed to ensure that real-time systems could operate safely, even in the face of faults or disruptions.

The importance of safety grew, particularly in critical applications such as transportation or medical devices. More recently, as real-time systems increasingly became interconnected and even accessible over the internet, security emerged as another crucial consideration. With the potential for cyber-attacks and unauthorized access, it became necessary to safeguard not only the data but also the integrity and availability of the communication system itself.

Today, a comprehensive real-time communication system must meet all three criteria: real-time responsiveness, safety, and security. It is no longer advisable to start from scratch when designing an embedded communication system for any real-time application. Once, it was quite common for developers to take an ad-hoc approach, such as repurposing one of the serial ports to share it among multiple nodes, effectively creating an RS485-style network. However, this approach does not accommodate the increasing complexity of real-time systems.

2.1 Is There a Best Fit?

In German, there's a saying "Es gibt keine eierlegende Wollmilchsau," which can be translated to "there is no one-size-fits-all solution" or, more literally, "there is no egg-laying wool-milk pig." This saying applies here as well. Regrettably, there is no single networking technology that is universally suitable for all applications. Each application has its unique set of requirements and constraints, making it necessary to carefully evaluate and select the appropriate communication technology and protocols. Therefore, it is essential to consider the specific needs of the application and match them with the most suitable networking technology available, taking into account factors such as required throughput, real-time responsiveness, safety, and security.

2.2 The Basics: How Much Data, How Often?

First, assess the overall architecture of your system. In addition to real-time requirements and the timeframe within which a complete control step must be executed, consider the total number of inputs and outputs required, their distances apart, and the number of signals and their data lengths that need to be exchanged within each timeframe and between devices. In general, it is not advisable to push any system "to its limits," so any networking technology you choose should have enough capability to accommodate your application's growth over time.

2.3 Are There Safety and Security Requirements?

Once you've established the applicable timeframe for your application, it is crucial to determine what safety and security measures are necessary. If your application must adhere to specific safety standards or certifications, your choices regarding communication networks will automatically narrow. For this article, we focus on the real-time requirements. When conducting your research, double-check the latest developments—all active fieldbus organizations and committees are continually working on improving both safety and security.

2.4 Are There Synchronization Requirements?

Consider whether any signals require synchronization, meaning that inputs should be captured at the same moment in time. Synchronization is critical for applications where multiple inputs are combined. In real-time communication systems, synchronization plays an important role in ensuring accurate data transmission and interpretation. Some applications demand synchronization due to their nature (e.g., syncing multiple manipulators working on the same material simultaneously), while other effects might be more subtle: Consider a scenario where an analog sensor generates input data every 100ms based on its internal timer. The

transmission of this data onto a network also occurs every 100ms, triggered by a separate network timer. If these timers are not synchronized, they may gradually drift apart, leading to two possible scenarios:

1. **Duplicate Data Transmission:** If the network timer's window is shorter than the sensor's, the sensor may not have generated new input data by the time the network is ready to transmit. In this case, the same data could be transmitted twice.

2. **Data Loss:** If the sensor's timer window is shorter than the network's, a new value may be generated before the previous one has been transmitted. This situation can lead to skipped or lost data.

The impact of these scenarios greatly depends on the signal and its usage. For instance, if the value represents temperature and the main processing unit only needs to know if it falls within the correct range, these scenarios have no effect. However, if it is a counter or a rapidly changing signal representing a wave, missing or duplicated data may have serious consequences.

2.5 Other Considerations

When selecting a real-time communication system, there are many additional considerations: Are off-the-shelf products, development, and diagnostic tools readily available? Can it easily integrate with existing (or planned) systems? If hard real-time of single milliseconds is a requirement, such integration may need to go "deeper" into a system, potentially requiring custom software at the lowest levels of the hardware.

2.6 Too Many Choices…

Understanding the specific requirements of your application—real-time responsiveness, safety, security, system architecture, and synchronization—can guide you in selecting a suitable communication network. If you start at zero, a potential starting point for gaining an overview of available fieldbusses is the Wikipedia entry titled "Fieldbus." However, note that this list captures only a fraction of the available fieldbusses. The domain of industrial communication networks is vast and continuously evolving, with many fieldbusses, some not even officially standardized. Beyond the widely-recognized fieldbusses, many networks, often crafted by manufacturers or consortia, cater to specific applications or industries. They might offer distinct features, specialized protocols, or proprietary technology tailored to certain application needs.

For instance, the Controller Area Network (CAN) is a versatile communication protocol supporting numerous applications through its specialized protocols. Protocols like J1939 cater to commercial vehicles (like construction, agriculture), standardizing message formatting and signaling to facilitate manufacturers in crafting interoperable components. NMEA 2000, by the National Marine Electronics Association (NMEA), aids the integration of marine electronics, streamlining the configuration and management of intricate marine systems. CANaerospace, designed for aerospace, meets the distinct demands of avionic systems, ensuring reliable data exchange in aircraft.

The CANopen protocol, with its flexibility, boasts many device and application profiles, such as those for elevators, emergency vehicles, and CleANopen for waste collection vehicles. These profiles determine the communication behavior and data structures for devices or entire applications, simplifying the development process.

Moving beyond CAN, some 10+ different solutions exist for Ethernet-capable real-time communication, each targeting varied applications. As a general rule, if your application's real-time requirement is roughly 100ms or more, you have a multitude of choices since most embedded communication networks or fieldbus can fulfil these demands, even for more extensive systems. However, for vast machinery (spanning several hundred meters of cable and beyond), scrutinizing communication runtime and throughput is essential.

For stringent real-time requirements, as short as 10ms or even less, it's imperative to diligently review which network technologies can satisfy your needs. Typically, a time-triggered communication system (available on CAN, Ethernet, and other platforms) is the most deterministic. Here, each signal with real-time requirements is allocated an exclusive timeslot, ensuring predictable signal transmission.

2.7 What's Next?

As an expert in CAN and CANopen communications, the next part III of this series will focus on CAN and CANopen as examples for the many embedded communication systems available. I will explore its suitability for diverse systems with real-time requirements, highlighting achievable response times, areas demanding meticulous attention, and situations that push boundaries, suggesting the evaluation of alternatives.

3 The Temporal Dynamics of CAN-Based Systems

After reviewing the basic requirements for selecting a real-time capable embedded communication system, I will now examine the real-time capabilities and limitations of CAN and CANopen in greater detail.

The Controller Area Network (CAN) protocol serves as the foundation for numerous applications across a wide range of industries, each with its own distinct real-time demands. Prominent examples like CANopen and J1939 highlight the diverse adaptations of this protocol to meet specific needs. It's important to note that the real-time requirements for these applications are not uniform across the board. While some applications require reaction times measured in milliseconds, many others operate effectively under more relaxed criteria. Factors such as physical constraints, network topologies, and computational tasks play a crucial role in shaping these requirements. As we explore tighter real-time constraints, the complexity of communication configurations and code handling increases. However, when real-time requirements are more relaxed, it opens up opportunities for simpler, more streamlined system designs without sacrificing functionality or reliability.

Although bothy safety and security have been addressed with (by CANopen Safety and CANcrypt) there is currently no standardized solution that provides both. The CiA (CAN in Automation) user's group currently has multiple working groups reviewing various aspects of both safe and secure communication with CAN, CAN FD and CAN XL.

3.1 Real-Time Capabilities of CAN

CAN's real-time effectiveness is closely tied to its communication speed, and further affected by its priority-based arbitration mechanism. Calculating CAN frame transmission times is not a straightforward task; the time depends on both the number of data bytes and their content. This complexity arises because stuff bits may be added to a frame depending on its data. Therefore, the following determined values should be considered as approximations, providing a general sense of the scope at hand.

It's important to remember that your maximum bitrate also depends on the physical topology of the cabling, and depending on your application, the total transfer required for a single control cycle might include two transmission paths: one for input data to the control unit and another from the control unit to the outputs.

Though I focus on CAN here, most of the following considerations also apply to the CAN FD (Flexible Data Rate) and CAN XL variants. Both of these protocols feature a dual bitrate mechanism, further enhancing their data throughput capabilities. However, when discussing timing-related dynamics, most of the considerations I have outlined predominantly apply to the "nominal bitrate." This foundational bitrate essentially establishes the pace for control information such as arbitration, acknowledgment, and error signaling. For those using CAN FD and CAN XL, it's crucial to be aware of the additional complexities introduced by the "data phase bitrate," which governs the transmission of the actual data. One of the key concerns in these systems is determining the maximum duration a lengthy message might occupy the bus and how much longer this delay might be compared to the longest classical CAN frame with 8 bytes.

At its maximum speed of 1Mbps, CAN allows for the exchange of more than ten frames within a millisecond. Conversely, at a modest rate of 125kbps, it averages around one frame per millisecond. Beyond mere transmission times, signals or frames can experience delays if higher-priority communication is in the queue. To put it simply, the worst-case transmission time would be the sum of the frame's own transmission time and the delay expected from the longest sequence of higher-priority traffic in the system. This assumes that all communication happens on a single CAN bus. If signals need to be forwarded via bridges or gateways, delays become longer and even more challenging to predict.

The system of message prioritization can be a double-edged sword. However, there is a mitigating factor: by strategically limiting the duration of sequential

high-priority traffic, even communications with the lowest priority can be dispatched with minimal delay. This approach ensures consistent and timely data exchange throughout the system.

Looking at CAN (and the FD and XL variants) by itself, it is clear that “as is” it is not deterministic. A single device producing high priority frames can block the communication for all others. To make CAN deterministic, we need to ensure a controlled frame triggering – when may which CAN ID be used. To activate CAN's real-time capabilities, consider the following design goals. While these guidelines may vary based on application specifics, they serve as a reliable starting point:

A) Aim to keep the overall busload at a level where even low-priority frames have sufficient time to access the bus. While the exact threshold can vary by application, my initial recommendation is to stay below 75% busload (less if communication is purely change-of-state-based).

B) Ensure that no individual node can generate an extended stream of consecutive messages. Some drivers offer a transmission "throttle" to limit the maximum transmission rate.

C) For those seeking finer control over transmission timing and sources, consider the SYNC mode of CANopen. This mode enables trigger messages, providing enhanced control over transmission schedules, allowing trigger modes like those used by time triggered systems.

3.2 Mastering the Temporal Dynamics of CAN-Based Systems

After exploring the various use cases and their respective temporal demands of CAN-based systems, you can imagine that matching CAN configurations to specific time requirements is both an art and a science. The following table summarizes some of the main numbers and factors to consider. The first section of the table gives you a summary of the CAN timings and throughputs that you can expect at various bitrates – this is all for classical CAN using 11bt CAN message identifiers. The fact that even at the lower bitrates we are still talking about potentially thousands of CAN frames per second never stops amazing me. There is sooo much room for communication that one can easily grasp that with some well-defined parameters on how to use all this “space” one can very well design real-time capable systems.

CAN Timings and Throughput				
CAN Bitrate	1000kbps	500kbps	250kbps	125kbps
Shortest 11bit CAN Frame	50us	100us	200us	400us
Longest 11bit CAN Frame	130us	260us	520us	1040us
Max shortest frames per second	20000	10000	5000	2500
Max longest frames pers second	7700	3850	1925	960
Max data throughput 70% load	400kbps	200kbps	100kbps	50kbps

Possible transmission delays (ms)				
CAN Bitrate	1000kbps	500kbps	250kbps	125kbps
Network in use	0.13	0.26	0.52	1.04
Arbitration delay for 5 frames	0.5	1	2	4
Transmission throttle delay	0.5	1	2	4
Sum	1.13	2.26	4.52	9.04

MCU/CPU Processing delays	
Higher Layer Stack Processing	fractions of milliseconds
App processing RTOS	fractions of milliseconds
OS (non RT) minimal proc. Delay	single milliseconds
OS (non RT) minor delay	some 50 to 100ms
OS (non RT) mayor delay	can be above 1s

TABLE: BALLPARK FIGURES FOR TRANSMISSION DELAYS

The next section of the table shows potential transmission delays and depends on many factors. Therefore, it is only a rough estimate for a specific use case, you need to adapt it to your own use case. The first row shows the delay even the highest priority will have, if the bus is currently in use (arbitration already started, transmitter is too late to join). Transmission has to wait, until the current frame completed. The second row shows an arbitration delay – if there are other devices also trying to transmit a frame, how long do we have to wait? Here we show the delay for 5 other frames currently pending for transmission and having a higher priority followed by a line of further delays, if a throttle mechanism is used protecting from back-2-back transmissions. Further on in this article we will review what can be done if the sum of delays shown is unacceptable in your application.

The last section of the table shows the potential processing delays caused by executing various code on the device handling the CAN communication. Here we assume that a modern 32bit MCU with integrated CAN interface is used running at 80Mhz or faster. In such environments, the code execution directly related to

handling the CAN frames is typically marginal. Potential delays come from “what else is happening” on that MCU.

Translating this knowledge into real-world system performance requires actionable strategies and considerations. With the previously established benchmarks from part I—seconds, 100ms, 10ms, and single milliseconds—as our guideposts, let's review practical recommendations for optimizing your CAN-based systems.

3.3 Mastering CAN Applications with Response Times Beyond Seconds

In the domain of applications operating with delays stretching into seconds or even minutes, designing CAN-based systems to meet these response times is not particularly challenging. Interestingly, even a device burdened by sub-optimal drivers or firmware might still be suitable, as even sub-optimal drivers will even still perform within 10 to 100 of milliseconds.

However, when working with devices that rely on non-real-time operating systems, the challenge lies not so much in countering communication delays, but rather in upholding consistent performance and avoiding the worst-case possible delay. Regular testing and thorough monitoring are essential to ensure that these devices never falter in allocating the necessary resources for seamless CAN communication. It is also crucial to proactively curb any potential system disruptions. Simple yet effective measures, such as ensuring the absence of updates or other resource-draining operations during active communication periods, can strengthen the system's responsiveness and reliability.

3.4 Mastering CAN Applications with Response Times of 100ms

This domain is where the potential of CAN, in synergy with higher-layer protocols like CANopen, truly comes into play. The CANopen PDO (Process Data Object)

communication mechanisms inherent in CANopen provide users with flexible control, simplifying the configuration of message content and triggering. These PDOs facilitate real-time data exchange between nodes, optimizing communication efficiency.

At this response time, CAN ID assignment and overall busload remain critical, but not overwhelmingly so, as they are unlikely to cause delays approaching anywhere near 100ms. The system architecture should be designed such that even messages with the lowest priority have timely bus access, ensuring their transmission within the stipulated timeframes. As we navigate this middle ground, it becomes increasingly important to review potential high-priority message bursts. Back-to-back high-priority transmissions can dominate the bus, posing risks of delays for lower-priority messages. Effective strategies for avoidance or control, such as limits on what each node can transmit per timeframe or synchronized triggering, can be employed to mitigate these bursts, ensuring more predictable and harmonious bus communication, even as the system scales.

While many non-real-time OSs can still achieve a 100ms response, it is advisable to lean towards an RTOS (Real-Time Operating System) in such scenarios (if an OS is required at all, many simple IO devices typically do not have an OS at all). Using an RTOS aligns naturally with the demands of a 100ms response window. If a non-RTOS is chosen, rigorous and extended testing becomes imperative to ensure the OS consistently meets the desired response times under all conceivable operational circumstances.

Within this 100ms response time framework, the software and firmware requirements remain relatively forgiving. Specific optimizations are often unnecessary; even drivers or stack implementations deemed sub-optimal in high-performance environments (for example not taking advantage of CAN controller hardware features for advanced filtering and buffering) can adequately serve the purpose.

3.5 Mastering CAN Applications with Response Times of 10ms

As we move into the 10ms response time zone, precision and control over every system component becomes essential. This is where detailed scrutiny of network data flow is essential.

Time-triggered networks, optimized for hard real-time applications, are often the preferred choice in such demanding scenarios. The CANopen SYNC mode is an effective approach to mimic communication behavior as used in time-triggered communication systems. By utilizing SYNC triggering messages, it enables specific nodes to transmit their associated PDO messages at precise moments, bringing predictability and consistency to system communication.

While a Real-Time Operating System (RTOS) might seem ideal for such tight timing requirements, it comes with its set of challenges. An RTOS offers a range of configuration options, and managing these tasks requires careful coordination. Within the tight 10ms window, the process involves a sensor sending its current data, a control device with an RTOS receiving and processing this data, and then acting upon it.

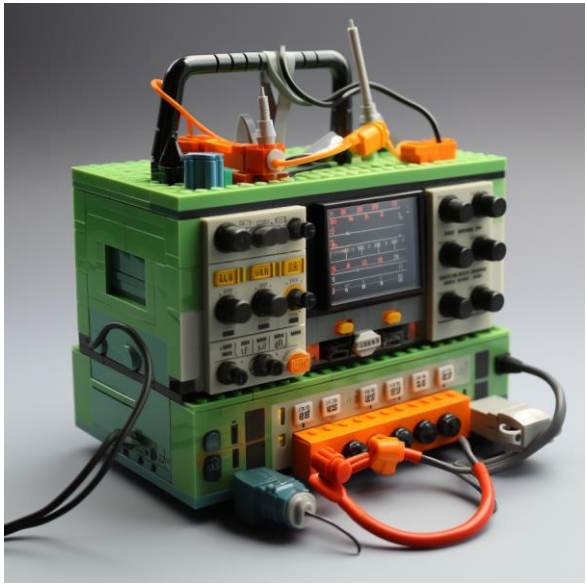
However, simply implementing an RTOS does not guarantee the desired outcomes. Task prioritization and configuration must align perfectly with the system's stringent timing requirements. Additionally, a detailed review of driver functionality, firmware, and stack structures is crucial. Potential issues, such as priority inversion where a low-priority message in the queue might delay a higher-priority one, need to be addressed. Highly optimized drivers can address priority inversion, but this may cause changes in transmission sequences. A change in transmission sequences can be problematic for certain higher-layer protocols and needs to be reviewed carefully.

3.6 Mastering CAN Applications with Response Times of 1ms

Venturing into the 1ms response time territory for CAN-based applications is akin to treading on the edge of the protocol's capabilities. These applications truly push the boundaries, requiring an unparalleled level of optimization and attention to every detail.

At this threshold, conventional approaches and tools often prove inadequate. Even some RTOSs, which typically excel in managing real-time tasks, may struggle to consistently adhere to this tight window. This necessitates reliance on microcontroller-specific implementations, where most tasks are handled directly within interrupt service routines, bypassing the typical layers of an RTOS.

The extreme precision required at this level means that many system configurations may need to be hard-coded, potentially bypassing higher-layer protocol stacks like CANopen that would otherwise delay processing. This also helps avoid potential delays introduced by configuration handling, ensuring maximum predictability. Every component, message, and byte transmitted on the network must be judiciously managed.



Given the stringent requirements, if a 1ms response time is a necessity for your application, it is wise to review if other communication solutions beyond CAN might be better suited to your needs. This domain requires significant commitment in terms of development time, testing, and optimization. If this endeavor is taken on, one should be fully prepared for a time-consuming project journey.

3.7 Concluding the Temporal Dynamics of CAN-Based Systems

The exploration of the temporal dynamics of CAN-based systems has underscored the adaptability and capabilities of the CAN protocol across various response time requirements. For applications with response times extending beyond seconds, there is less emphasis on precise timing, and decisions regarding CAN ID usage, higher-layer protocols employed, or the operating system selected generally have a less pronounced impact on performance.

However, as we reach into tighter time constraints of 100ms and 10ms, system design considerations become of greatest importance. These include total bus load, message priorities, and the strategic employment of functionalities like CANopen's SYNC mode. When navigating the demanding 1ms response time domain, every element of the system requires meticulous attention and may even prompt a re-evaluation of the network system selected.

In conclusion, understanding the balance between application requirements, the inherent strengths of CAN, and the related temporal constraints is vital. It's this knowledge that empowers CAN system designers to make informed decisions across diverse temporal scenarios.

In the next and last article of this series, we will go deep into the technical details, examining the configuration and optimization options available with CANopen source code solutions, such as Micro CANopen Plus. We will provide practical insights into how the inherent strengths of CANopen can be harnessed to meet a broad range of real-time application demands. This final part will offer readers a tangible guide to optimizing real-world CANopen implementations for shortest processing times.

4 From Theory to Practice: CANopen Source Code Configuration

As we have seen in the previous parts of this series, the adaptability and fine-tuning of CANopen systems play a crucial role in meeting real-time application demands. In this final instalment of our series, I show you technical details of CANopen source code configuration, shedding light on the various ways it can be optimized for efficient real-time performance.

Throughout this article, I will examine specific examples using EmSA's Micro CANopen Plus source code. These examples will illuminate the process of configuring, optimizing, and fine-tuning a CANopen stack to cater to advanced temporal requirements and possible system constraints. While my focus is on Micro CANopen Plus, it's worth noting that the principles and methods I will explore here are likely to work with other CANopen source code implementations in a similar way.

Whether you are an experienced CAN system designer looking to sharpen your optimization skills or a newcomer seeking to understand the nuances of real-time

CANopen configuration, this part aims to provide comprehensive insights and guidance, translating the theoretical knowledge we gained into practical implementation.

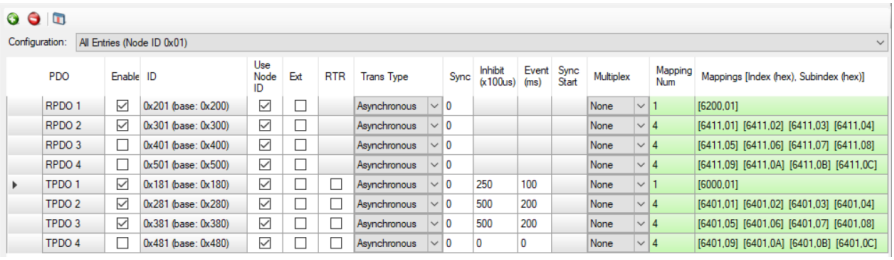
4.1 Different CANopen PDO Configurations and their Impact on Response Time

The configuration of CANopen PDOs (Process Data Objects) plays a critical role in determining the response time in various applications. Depending on the required response time and the necessity to synchronize signals across multiple nodes, different PDO triggering mechanisms can be applied.

4.1.1 Response Time of 100ms

For applications where the required response time is 100ms or longer, there are typically two configuration methods that work well (and can also be combined):

- **PDO Triggering by Event Time (Cyclic Transmission):** Here, the PDOs are transmitted cyclically at specified time intervals, such as every 50ms. This periodic transmission ensures consistent response times.
- **Change of State (COS) Detection with Inhibit Time:** This configuration transmits PDOs based on changes in state, with a minimum time (inhibit time) between transmissions. This inhibit time ensures that a toggling input does not produce back to back messages.



Configuration: All Entries (Node ID 0x01)

	PDO	Enable	ID	Use Node ID	Ext	RTR	Trans Type	Sync	Inhibit (x100us)	Event (ms)	Sync Start	Multiplex	Mapping Num	Mappings [Index (hex), Subindex (hex)]
	RPDO 1	☑	0x201 (base: 0x200)	☑	☐		Asynchronous	0				None	1	[6200,01]
	RPDO 2	☑	0x301 (base: 0x300)	☑	☐		Asynchronous	0				None	4	[6411,01] [6411,02] [6411,03] [6411,04]
	RPDO 3	☐	0x401 (base: 0x400)	☑	☐		Asynchronous	0				None	4	[6411,05] [6411,06] [6411,07] [6411,08]
	RPDO 4	☐	0x501 (base: 0x500)	☑	☐		Asynchronous	0				None	4	[6411,09] [6411,0A] [6411,0B] [6411,0C]
▸	TPDO 1	☑	0x181 (base: 0x180)	☑	☐	☐	Asynchronous	0	250	100		None	1	[6000,01]
	TPDO 2	☑	0x281 (base: 0x280)	☑	☐	☐	Asynchronous	0	500	200		None	4	[6401,01] [6401,02] [6401,03] [6401,04]
	TPDO 3	☑	0x381 (base: 0x380)	☑	☐	☐	Asynchronous	0	500	200		None	4	[6401,05] [6401,06] [6401,07] [6401,08]
	TPDO 4	☐	0x481 (base: 0x480)	☑	☐	☐	Asynchronous	0	0	0		None	4	[6401,09] [6401,0A] [6401,0B] [6401,0C]

PDO Configuration with CANopen Architect

4.1.2 Smaller Response Times or Synchronized Signals Across Multiple Nodes

For applications requiring smaller response times or where there is a need to synchronize signals across multiple nodes, the SYNC mode becomes the preferred method:

- **SYNC Mode:** In this configuration, one SYNC producer generates a SYNC CANopen message at stable, repeating intervals, such as every 10ms. This SYNC message serves as a triggering mechanism, used by all devices in the network to apply data synchronously, at the same time.

4.1.3 Advanced SYNC usage

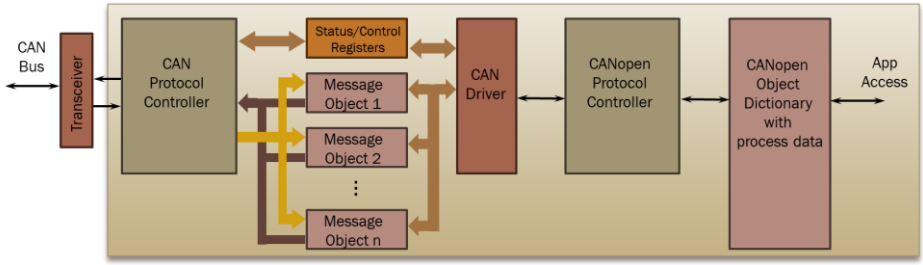
When using the CANopen SYNC mode, there are two advanced features that you can take advantage of. First, most SYNC consumers allow the configuration of the SYNC CAN message identifier to be used. So you could configure a system to use multiple SYNC trigger messages and select which devices react to which trigger. Secondly, the latest version of CANopen supports the use of SYNC with an integrated configurable counter. As an example, this could be configured to count until 4. On the SYNC consumer side, you can configure the count value that each consumer listens to, again providing the option of grouping devices to react on specific SYNCs on the system.

Choosing the right PDO configuration is vital for achieving the lowest response times. While cyclic transmission and COS detection with inhibit time are suitable for more relaxed response time requirements, the SYNC mode becomes essential when handling tighter time constraints or needing to synchronize multiple devices. For more details on these different trigger mechanisms and how they can be combined, see our video https://www.youtube.com/watch?v=vxi5awte5eo

4.2 Generic Data flow in a CANopen Protocol Stack

On the lowest hardware level, a CAN controller will be receiving CAN frames. Depending on filters, these might be placed into pre-selected buffers or queues, and an interrupt signal will be generated. The processor handling the CAN controller starts processing the "CAN receive interrupt service" – typically part of a processor-specific driver. A generic driver will now simply pass on the CAN data to another software queue for later processing; I will discuss advanced options later.

At any time, the application program might update some of the process data to be transmitted via CANopen.



To keep the CANopen stack alive, there will be a function, such as "ProcessStack()", that needs to be called frequently (for example, simply in a "main while(1) background loop"). When called, this function typically first checks if CANopen messages were received; if so, they are processed. If the data involves updating process data, then there is typically a callback to inform the application that new process data has arrived.

When all received CANopen messages are processed, the function checks if there is anything to transmit. It may detect that outgoing process data was modified by the application, and depending on the configuration of timers and transmission mode, initiate the transmission of a corresponding CANopen message.

Such transmissions are typically passed to the driver level, possibly into a transmit queue, and it depends on the driver configuration when exactly this CANopen message will be passed to the CAN controller for transmission.

4.3 Basic configuration and control options

Unless the required processing and response time is smaller than 100ms such a data flow works good enough for most applications. If required response times get smaller, you should start looking into possible optimizations. When reviewing the generic data flow above, possible optimizations include:

4.3.1 CAN Driver Optimization for Receive

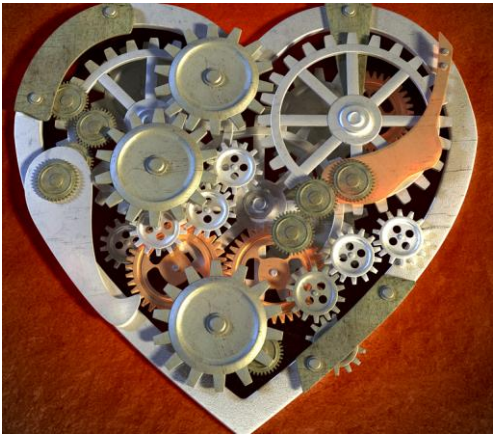
Many default drivers supplied by chip manufacturers (or even CANopen stack providers) might not take full advantage of the specific features of a CAN controller. One of the first possible optimization checks is to ensure that where possible, hardware receive filtering and hardware receive buffers or queues are utilized, thus eliminating the need for a long (delaying) software receive queue.

4.3.2 CAN Driver Optimization for Transmit

Before reviewing this, consider what is more critical to your application – whether this device can transmit as many CAN frames back-to-back as possible, or whether the transmission should be somewhat throttled to ensure that no single device can produce too long a period of high-priority back-to-back traffic. If it should be throttled, consider implementing a transmit trigger based on a timer, such as being able to transmit one CAN frame every millisecond (our default driver uses the 1ms timer interrupt for this).

4.3.3 Considerations for "ProcessStack()"

A typical question regarding "ProcessStack()" is how often it should be called and what the worst-case execution time is. Some prefer to call it from a fixed timer interrupt instead of the background loop. There is no generic answer to these questions. In our Micro CANopen Plus implementation, we try to keep the execution time short by NOT executing all pending CANopen tasks but only the most vital ones. How often it should be called depends heavily on the local device's communication. Our Micro CANopen Plus implementation, however, has a slick feature here: with every call, only the most critical pending tasks of the CANopen stack are performed. Producing the heartbeat message is always the least important task. Therefore, by monitoring the device's heartbeat signal's accuracy, you can determine if calls to "ProcessStack()" occur often enough. If there are not enough calls to "ProcessStack()", the heartbeat becomes slower than specified, or it may not be transmitted at all.



4.3.4 Note on Return Value

Another important factor is the function's return value. It returns TRUE when a pending CANopen task was executed, and FALSE when there is no CANopen task pending. If you want to ensure that all pending CANopen tasks are executed in your code, simply use:

```
while(ProcessStack())
{
}
```

This will keep re-calling the function until all pending CANopen tasks have been executed.

4.3.5 Direct Task Trigger

The function "ProcessStack()" serves those who prefer not to go into the details of all the CANopen tasks executed from within. For further optimization, an application can bypass calling this function and directly invoke the dedicated CANopen tasks: "ProcessStackRx()" and "ProcessStackTick()".

- **Sub-task "ProcessStackRx()":** This task handles processing a received CANopen message. For an optimized call, it would ideally be initiated directly from the CAN receive interrupt or triggered by some signal set in the CAN receive interrupt.
- **Sub-task "ProcessStackTick()":** This task checks if the process data to be transmitted has changed (or was triggered for transmission) and if any actions based on the millisecond timer need to be taken. The most efficient way to call this is only after process data has changed or the millisecond timer has incremented.

This approach provides a more refined control over the execution of specific tasks within the CANopen stack, allowing for more precise tuning of performance and responsiveness.

4.4 Bringing together CAN Driver, CANopen Stack and Application

On most 32-bit-based microcontrollers, the enhancements discussed so far are suitable for bringing the total response time down to a range of 10ms to "a few milliseconds." This can be achieved without requiring optimizations that lead to a fully custom implementation that might be challenging to maintain. These optimizations were confined to leveraging individually triggered CANopen stack processes when needed.

In general, this can be taken further. However, making changes at such an intrinsic level in a system can make it much more challenging to maintain or port to a different architecture when necessary. Therefore, the following is more to

illustrate "what is theoretically possible," pushing optimizations beyond the point where a system remains easy to test, maintain, and port.

On the lowest hardware level, review if your CAN controller is configured to directly create a CAN receive interrupt with the reception of the SYNC message and if you can easily detect the difference to any other CANopen message (e.g. own filter/receive buffer).

The only reasonable CANopen PDO communication mode for further real-time improvements would be the CANopen SYNC mode. If that is used and we concentrate on optimizing it, then the previous other optimizations might become redundant.

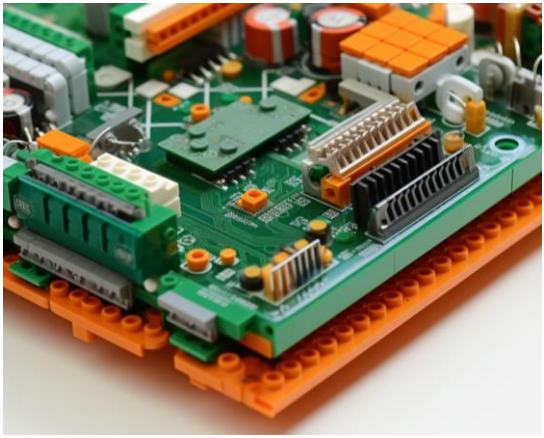
Focusing on SYNC optimization would require us to modify the CAN receive interrupt service routine to directly call the CANopen stack function(s) responsible for SYNC handling when a SYNC signal is received. In the case of Micro CANopen Plus, this would be the function “HandleSync()”. When executing this from within the interrupt service routine, please keep in mind:

- Not to store this SYNC in the regular receive queue (we already process it).
- That for both SYNC-related transmit and receive data, call back functions to the application will be called—still executing at the interrupt service level.
- When using an RTOS, a better solution would be to set a trigger signal in the interrupt that SYNC was received, subsequently triggering the execution task immediately after the interrupt has completed.

With such a modification, a response time within a millisecond is achievable. If all devices participating in the SYNC communication implement the SYNC handling with equal optimization, the variation among the devices (e.g. when they each apply their outputs synchronously) can be as low as a few microseconds.

Nevertheless, these are extreme values that have been observed to work in test and lab environments. For real-world applications demanding such short response or sync times, careful testing would be required to ensure that these targets can be reached under all realistic circumstances.

4.5 Final Conclusion: Navigating Complexity through Strategic Choices



The journey through this four-part series has provided an in-depth exploration of the components that shape embedded real-time communication. From hardware selection to advanced optimization techniques, the underlying theme that resonates is the pivotal role of response times in determining every aspect of system design and configuration.

1. **Hardware Selection**: The required response times dictate the hardware capabilities needed, influencing decisions on modules, possibly microcontrollers and other essential components.

2. **Operating System Considerations**: Whether working with an RTOS or implementing a more specific, bespoke system, the response times heavily influence how the operating system needs to be configured.

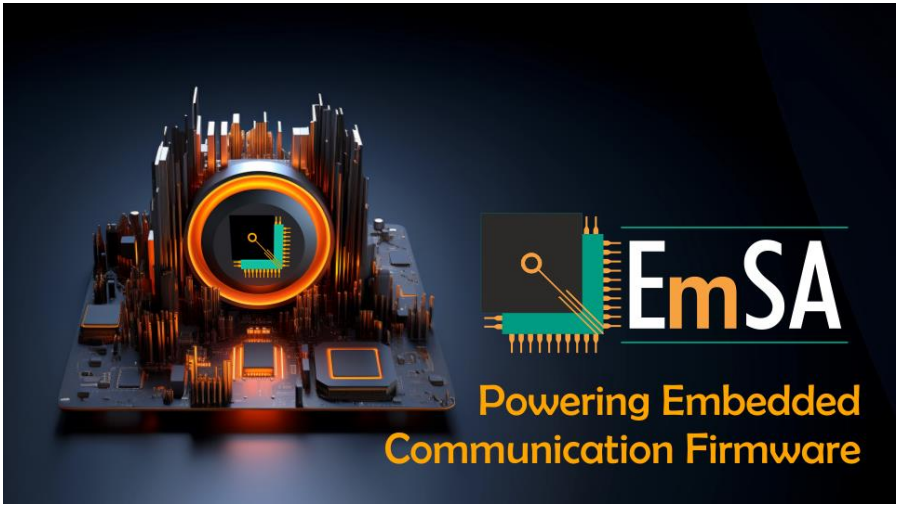
3. **Network Technologies**: Depending on the required throughput and speed, different network protocols and technologies must be taken into consideration. As an example, this series looked at the specifics of CANopen and its configurations, illustrating the nuanced choices required to meet different application demands.

4. **Optimization Choices**: Perhaps one of the most profound insights is the realization that optimization is not a one-size-fits-all approach. Depending on the required response times, certain optimizations become essential, while others can be bypassed. It's a matter of fine-tuning, understanding what needs to be harnessed, and what can be left untouched without affecting performance.

5. **Strategic Ignorance**: Contrary to the instinct to utilize every possible advantage, there are instances where the time frame allows for the

deliberate ignoring of certain optimizations. Not every register provided by a network controller needs to be exploited; it's a balance between performance and the demands of the particular application.

Through this series, I have illuminated the complex interplay of hardware, operating systems, and network technologies, all governed by the essential factor of response times. The insights offered serve as a guide for making strategic choices in system design, highlighting the importance of tailored optimization and thoughtful decision-making. These principles enable you to craft robust and efficient real-time communication systems, suited to your application demands.



5 Developing and Testing Real-Time CANopen solutions

In this chapter, we will introduce you to the functionalities and utilities of EmSA's comprehensive range of products in the context of real-time CANopen and CANopen FD applications. We offer various utilities and tools for configuration and analysis as well as hardware and software:

- **Configuring and Diagnosing Real-Time Behavior:**
  EmSA's sophisticated diagnostic tools aid you in configuring and diagnosing CANopen systems for their specific real-time behavior.

- **Implementing CANopen Nodes with Specific Real-Time Requirements:**
  Using our CANopen hardware modules, you can directly design I/O modules with basic digital or analog inputs and outputs. Our CANopen source code products can be used to implement customized CANopen devices.

5.1 Context of total Real-Time response times

For all timings and measurements further down, you have to keep in mind that when monitoring CANopen communication and doing a timing analysis based on these recordings, you only see a part of the total picture. We can measure the accuracy of a cycle time for transmission – but we do not know how long it took to

process that data internally. Specifically, the time for input data changing until a CANopen message is triggered is unknown.

If you need this timing very accurately, then an oscilloscope with CAN interpretation works well. Ensure that you are monitoring the input signal and the CAN lines. On signal change on the input you can then measure the time until you see the corresponding PDO.

Another alternative to get an approximation of total processing times is to connect the inputs and outputs of a single CANopen device to each other. You can then trigger the PDO to the output and wait for the “responding” PDO that contains the input data triggered by the change of that output. The timing between these two PDOs can give you an estimation of the processing times involved.

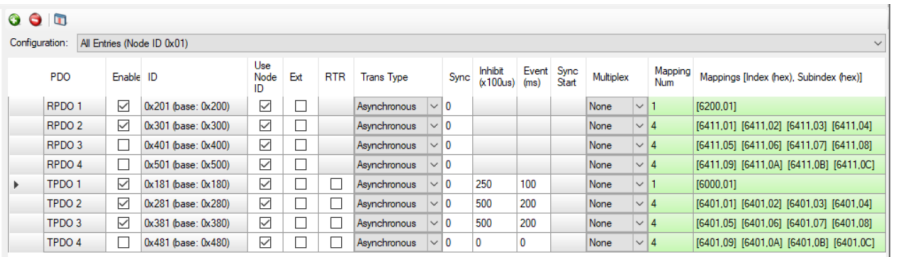
5.2 CANopen Architect: Managing CANopen Configurations

In CANopen, real-time related communication will be PDO based. No matter if cyclic or synced PDOs are used, CANopen Architect product allows you to quickly generate and modify PDO configurations.

At the heart of CANopen based real-time communication, PDOs (Process Data Objects) are used, which serve as the cornerstone for transferring data quickly and efficiently within a CANopen network.

The CANopen Architect stands as a central tool in managing these configurations. Whether you are working with event-driven, cyclic or synced PDOs, this tool facilitates quick generation and modification of PDO configurations.

For each device, all PDO configurations can be made based on our PDO configuration table as shown in the screen shot. This table provides quick access to all configurable PDO communication and mapping parameters.

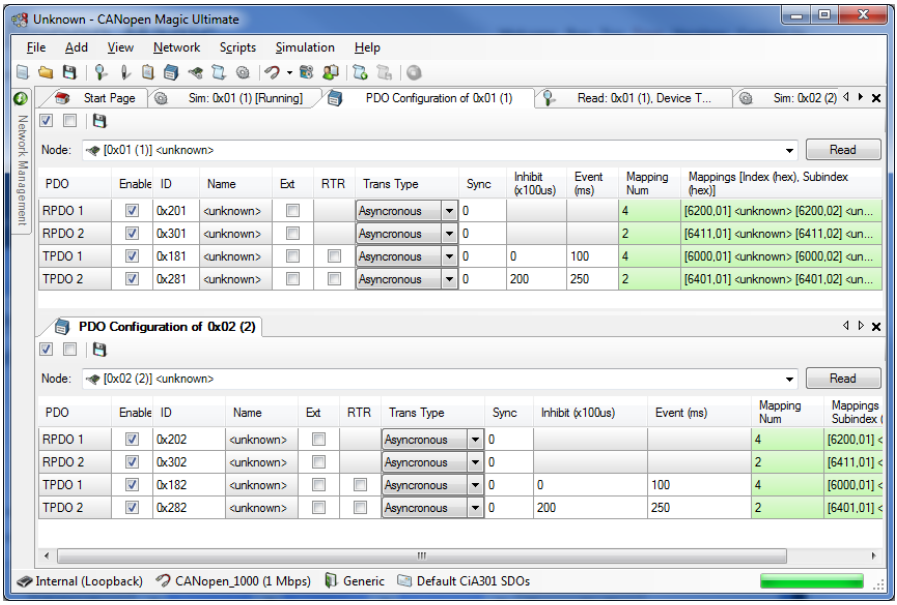

Configuration: All Entries (Node ID 0x01)

PDO	Enable	ID	Use Node ID	Ext	RTR	Trans Type	Sync	Inhibit (x100us)	Event (ms)	Sync Start	Multiplex	Mapping Num	Mappings [Index (hex), Subindex (hex)]
RPDO 1	☑	0x201 (base: 0x200)	☑	☐		Asynchronous	0				None	1	[6200,01]
RPDO 2	☑	0x301 (base: 0x300)	☑	☐		Asynchronous	0				None	4	[6411,01] [6411,02] [6411,03] [6411,04]
RPDO 3	☐	0x401 (base: 0x400)	☑	☐		Asynchronous	0				None	4	[6411,05] [6411,06] [6411,07] [6411,08]
RPDO 4	☐	0x501 (base: 0x500)	☑	☐		Asynchronous	0				None	4	[6411,09] [6411,0A] [6411,0B] [6411,0C]
TPDO 1	☑	0x181 (base: 0x180)	☑	☐	☐	Asynchronous	0	250	100		None	1	[6000,01]
TPDO 2	☑	0x281 (base: 0x280)	☑	☐	☐	Asynchronous	0	500	200		None	4	[6401,01] [6401,02] [6401,03] [6401,04]
TPDO 3	☑	0x381 (base: 0x380)	☑	☐	☐	Asynchronous	0	500	200		None	4	[6401,05] [6401,06] [6401,07] [6401,08]
TPDO 4	☐	0x481 (base: 0x480)	☑	☐	☐	Asynchronous	0	0	0		None	4	[6401,09] [6401,0A] [6401,0B] [6401,0C]

PDO CONFIGURATION WITH CANOPEN ARCHITECT

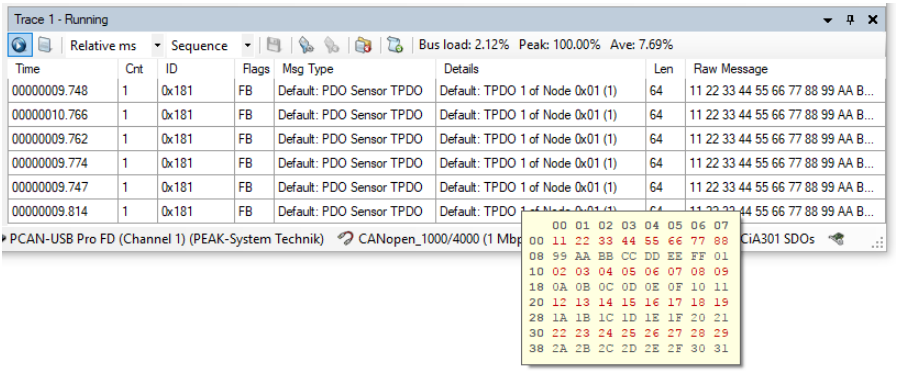
5.3 CANopen Magic: Loading and Testing CANopen Configurations

CANopen Magic serves as a vital utility in the CANopen environment, not only for loading of configurations into devices, but also for quickly testing single parameters, without the need for a complete re-configuration. Configurations can be verified quickly based on the same PDO configuration table used by CANopen Architect, giving you “live” access to these configurations.



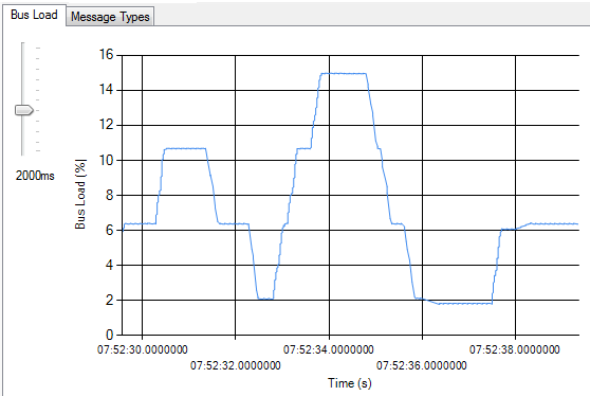
PDO Configuration Views

In addition, CANopen magic provides the ability to monitor and record all live CANopen traffic, including a high-resolution one-microsecond timestamp. This functionality ensures precise monitoring and analysis, allowing you to verify if response times are within the require time windows.



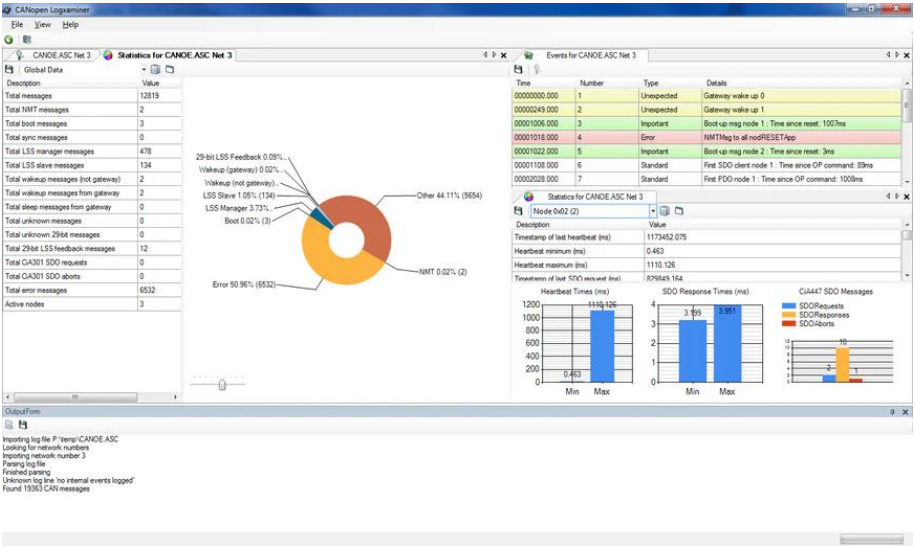
LIVE TRACE: HERE PDOS AND TIMING

When the configuration information provided includes details about the individual signals mapped into PDOs, a graph can be generate conating a graphical representation of the selected signal, including timing information (when the signal changed).



5.4 CANopen LogXaminer: Long-term Analysis

The CANopen LogXaminer is an indispensable tool for analyzing long log files or trace recordings (files with more than 1 million entries get broken up). Its core strength lies in the analysis of cycle and response times in depth. It can accurately identify the minimum and maximum cycle times for PDOs and Heartbeats, and similarly, it pinpoints the minimum and maximum response times for SDOs. The timer resolution depends on the resolution of the timer used by the utility generating the log file in the first place. Ensure that this resolution is good enough for your requirements. Good tools provide a timestamp based on Microseconds.



CANopen LogXaminer Analysis

In summary, CANopen LogXaminer specifically helps revealing the worst-case timing scenarios happening in the duration of the log file. Results are summarized for each node (only reviewing the TPDOs and SDO responses generated by that node).

5.5 CANopenIA Modules: Basic Profile CANopen devices

The CANopenIA System on Modules (SoM) is a robust solution engineered to facilitate the development of efficient real-time capable CANopen I/O nodes. Its compact design of flexible and responsive I/O nodes, significantly reducing the development time while ensuring reliable performance. Internal processing times for digital I/O are down to 15 Microseconds. Depending on configuration, the module is well suited to fulfil also real-time demands going down to single milliseconds.

5.6 Micro CANopen Source Code: Custom CANopen devices

Our Micro CANopen source code is well suited for implementing real-time capable CANopen devices, specific setting examples were listed in the previous chapter. Although this CANopen stack implementation is also available as a library, only the source code version offers all optimization offers mentioned.

Using this on an Arm Cortex-M microcontroller running at 80 Mhz or something with similar performance allows you to build real-time capable CANopen nodes.